Efficient Use of 3D Environment Models for Mobile Robot Simulation and Localization

Andreu Corominas Murtra, Eduard Trulls,
Josep M. Mirats Tur, Alberto Sanfeliu

Institut de Robòtica i Informàtica Industrial
C/Llorens i Artigas 4-6. 08028. Barcelona. Spain
{acorominas,etrulls,jmirats,asanfeliu}@iri.upc.edu
http://www.iri.upc.edu

Abstract. This paper provides a detailed description of a set of algorithms to efficiently manipulate 3D geometric models to compute physical constraints and range observation models, data that is usually required in real-time mobile robotics or simulation. Our approach uses a standard file format to describe the environment and processes the model using the *openGL* library, a widely-used programming interface for 3D scene manipulation. The paper also presents results on a test model for benchmarking, and on a model of a real urban environment, where the algorithms have been effectively used for real-time localization in a large urban setting.

Keywords: 3D environment models, observation models, mobile robot, 3D localization, simulation, openGL.

1 Introduction

For both simulation or real-time map-based localization, the mobile robotics community needs to implement the computation of expected environment constraints and expected sensor observations, the latter also called sensor models. For all these computations, an environment model is required and its fast and accurate manipulation is a key point for successful results of upper-level applications. This is even more important since current trends on mobile robotics leads to 3D environment models, which are richer descriptions of the reality but harder models to process.

Using computer accelerated graphics card for fast manipulation of 3D scenes has been addressed by robotic researchers from some yars ago, specially in the simulation domain [4, 2, 1], but it remains less explored in real-time localization. In real-time particle filter localization [8, 3], expected environment constraints and expected observations are computed massively at each iteration from each particle position, causing the software modules in charge of such computations being a key piece for the final success of the whole system. Unlike simulation

field, there exists few work reporting the use of accelerated graphics cards to implement the online computation of expected observations [5].

Moreover, detailed descriptions on processing 3D geometric models to efficiently compute physical constraints or expected observations are not common in the robotics literature. 3D geometric models have started to become an essential part of mobile robot environment models in recent years, specially for applications targeting outdoor urban environments, and thanks in part to the availability of powerful and affordable graphics cards on home computers. This paper wants to remedy the lack of a detailed description on how to efficiently manipulate such models, presenting in a detailed way the algorithms for an optimal use of the graphics card acceleration through the openGL library [7]. Thanks to their fast computation, the presented algorithms are successfully used for real-time map-based localization in a large urban pedestrian environment, demonstrating the potential of our implementation.

The paper begins introducing the 3D environment model in section 2. Section 3 details the algorithms to compute the physical constraints of wheeled platforms within the environment. Section 4 presents the algorithm to compute expected range observations, minimizing the computation time while keeping sensor's accuracy. Section 5 evaluates the performance of the algorithms and briefly describes a real application example. Finally, section 6 concludes the work.

2 Overview of the 3D Environment Model

The goal of our environment model is to represent the 3D geometry of a real, urban pedestrian environment in a useful way for map-based localization and simulation. Thus, the model incorporates the most static part of the environment, such as buildings, floor, benchs, and other fixed urban furniture.

Our approach uses the *.obj* geometry definition file format [6]. Originally developed for 3D computer animation, it has become an open format and a *de facto* exchange standard. We use two different maps for our intended application. One map features the full environment, $\mathcal{M}$, while the other consists only of those surfaces traversable by the robot, $\mathcal{M}_{floor}$, leaving holes instead of modelling obstacles. Figure 1 shows a view of the full model for the UPC (Universitat Politècnica de Catalunya) campus area and figure 2 shows real pictures of this environment and their approximate corresponding views in the 3D model.

This environment model implicity defines a coordinate frame in which all the geometry is referenced. Therefore a 3D position can be expressed with respect to the map frame as follows:

$$X_p^M = (x_p^M, y_p^M, z_p^M, \theta_p^M, \phi_p^M, \psi_p^M) \tag{1}$$

where the xyz coordinates define the *location* of the position and the $\theta\phi\psi$ coordinates parametrize the position *attitude* by means of the three Euler angles heading, pitch and roll.

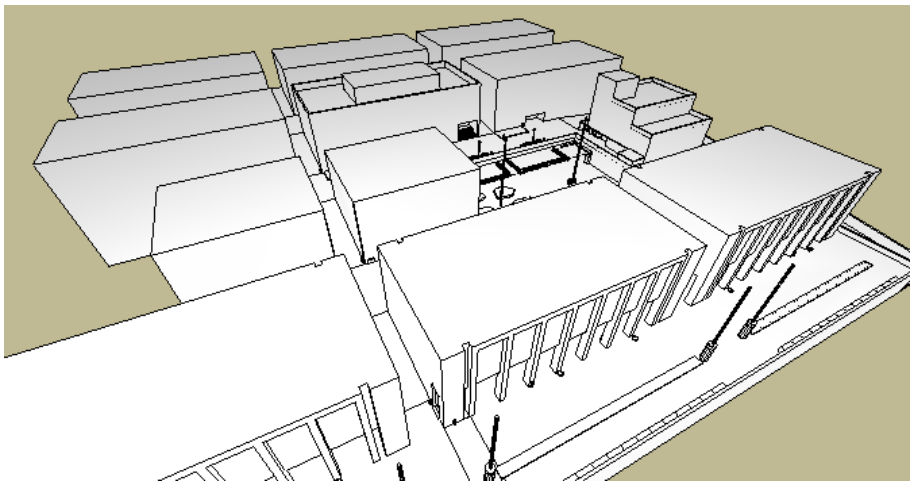

Fig. 1. 3D environment model of the UPC campus area.

Fig. 2. Pictures of the UPC campus environment and their approximate corresponding views of the 3D map. The 3D map only models the most static part of the environment.

3 Physical Constraints

Terrestrial, mobile robots have position constraints, due to gravity and the environment. Computing such physical constraints is a basic requirement for sim-

ulation but also for map-based localization, since the search space is reduced dramatically, therefore improving the performance of the localization algorithm.

3.1 Gravity constraints

A wheeled robot will always lie on the floor, due to gravity. For relatively slow platforms in can be assumed as well that the whole platform is a rigid body, so that a suspension system, if present, does not modify the attitude of the vehicle. With these assumptions, gravity constraints for height, pitch and roll can be derived from the environment.

The height constraint sets a height, z^M, for a given coordinate pair (x^M, y^M). To compute it only the floor map is used. Algorithm 1 outlines the algorithm. The key idea is to renderize the 3D model from an overhead view point, setting a projection that limits the rendering volume in depth and aperture in order to renderize only the relevant part of the model. Afterwards, by means of an openGL routine, the algorithm reads the depth of the central pixel.

Algorithm 1 Height constraint algorithm for 3D models

INPUT: $\mathcal{M}_{floor}, (x^M, y^M)$
OUTPUT: g_z

$setWindowSize(5,5)$; //sets rendering window size to 5x5 pixels
$setProjection(1^\circ, 1, z_{min}, z_{max})$ //1° of aperture, aspect ratio, depth limits
$X_{overhead} = (x^M, y^M, z^M_{overhead}, 0, \pi/2, 0)$; //sets an overhead view point, pitch= $\pi/2$
$renderUpdate(\mathcal{M}_{floor}, X_{overhead})$; //renders the model from $X_{overhead}$
$r = readZbuffer(3,3)$; //reads depth of central pixel
$g_z = z^M_{overhead} - r$;
return g_z;

The pitch constraint fixes the pitch variable of the platform to a given coordinate triplet $(x^M_p, y^M_p, \theta^M_p)$. The algorithm to compute the pitch constraint is outlined in algorithm 2. It employs the previous height constraint algorithm to compute the floor model's difference in height between the platform's frontmost and backmost points, g_{zf} and g_{zb}. The pitch angle is then computed using trivial trigonometry, while L is the distance between the above mentioned platform points. The roll constraint can be found in a similar way but computing the height constraint for the leftmost and rightmost platform points. Please note that the roll constraint applies to all wheeled platforms, but the pitch constraint does not apply to two-wheeled, self-balancing robots, such as ours, based on Segway platforms.

3.2 Offline height map computation

The constraints presented in the previous section are computed intensively in filtering applications such as map-based localization. To speed up online computations during real-time execution, a height grid is computed offline for the floor

Algorithm 2 Pitch constraint algorithm for 3D models

INPUT: $\mathcal{M}_{floor}, L, (x_p^M, y_p^M, \theta_p^M)$
OUTPUT: g_ϕ
$x_f^M = x_p^M + \frac{L}{2}\cos\theta_p^M$; //compute the platform's frontmost point
$y_f^M = y_p^M + \frac{L}{2}\sin\theta_p^M$; //likewise
$g_{zf} = heightConstraint(x_f^M, y_f^M)$; //compute height at frontmost point
$x_b^M = x_p^M - \frac{L}{2}\cos\theta_p^M$; //compute the platform's backmost point
$y_b^M = y_p^M - \frac{L}{2}\sin\theta_p^M$; //likewise
$g_{zb} = heightConstraint(x_b^M, y_b^M)$; //compute height at backmost point
$g_\phi = atan2(g_{zf} - g_{zb}, L)$;
return g_ϕ;

map. G_{height} is then a grid containing the height value z^M for pairs (x^M, y^M), thus being a discrete representation of the height constraint with a xy step γ:

$$G_{height}(i,j) = g_z(x_p^M, y_p^M) \quad | \quad i = (int)\frac{x_p^M - x_0^M}{\gamma},\ j = (int)\frac{y_p^M - y_0^M}{\gamma}, \quad (2)$$

where x_0^M and y_0^M are the map origin xy coordinates. The z^M value is computed offline by means of the height constraint (algorithm 1) along the grid points. Figure 3 shows the height grid for the UPC campus environment.

Note that this approach is valid for maps with a single traversable z-level, such as ours, and while our algorithms can be directly applied to multi-level maps further work would be required in determining the appropiate map section to compute. Since computing the pitch and roll constraints requires several z^M computations, the height grid speeds up these procedures as well. To avoid discretization problems, specially when computing pitch and roll constraints using G_{height}, we use lineal interpolation on the grid. Algorithm 3 summarizes the grid version of the height constraint.

Algorithm 3 Grid version of the height constraint

INPUT: $G_{height}, (x_p^M, y_p^M)$
OUTPUT: g_z
$i = (int)\frac{x_p^M - x_0^M}{\gamma};\ j = (int)\frac{y_p^M - y_0^M}{\gamma}$
$z_1 = G_{height}(i,j)$;
$(i_2, j_2) = nearestDiagonalGridIndex()$; //$i_2 = i \pm 1;\ j_2 = j \pm 1$;
$z_2 = G_{height}(i_2, j)$; //height of a neighbour cell
$z_3 = G_{height}(i, j_2)$; //height of a neighbour cell
$z_4 = G_{height}(i_2, j_2)$; //height of a neighbour cell
$g_z = interpolation(z_1, z_2, z_3, z_4, x_p^M, y_p^M)$;
return g_z;

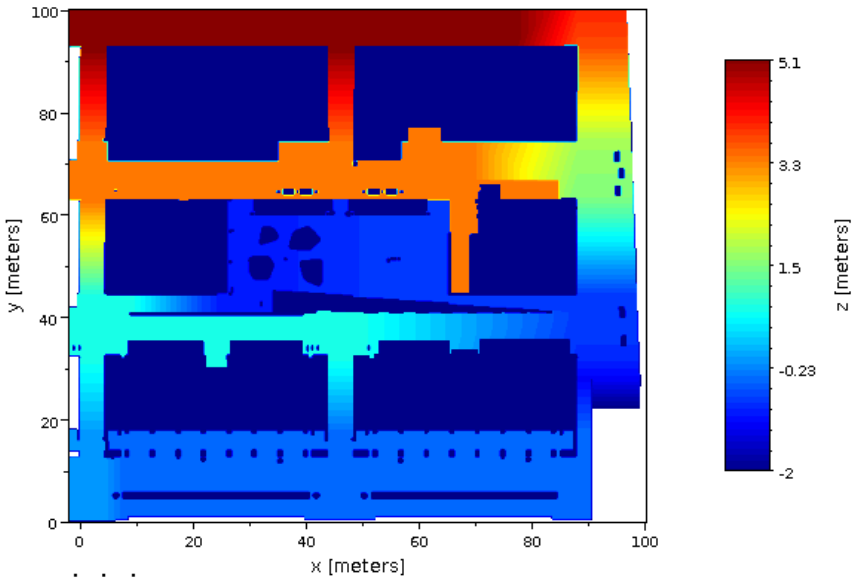

Fig. 3. Height grid, G_{height}, of the UPC campus environment.

4 Range Observation Model

Another key factor in dealing with environment models is the computation of expected observations, also called sensor models, or simulated sensors. This is also useful for both simulation and real-time map-based localization. This section outlines how, from a given 3D position in the environment, expected 2D range scans and expected 3D point clouds are computed. A common problem in either case is the computation of range data given the sensor position and a set of sensor parameters like angular aperture, number of points and angular accuracy. To compute these observation models, we use again openGL renderization and depth buffer reading, but the approach specially focuses on minimizing the computation time without violating sensor's accuracy and resolution. This minimization is achieved by reducing the rendering window size and the renderig volume as much as possible, while keeping the sensor accuracy.

The goal of a range observation model is to find a matrix $\mathbf{R}$ of ranges for a given sensor position X_s^M. Each element r_{ij} of the matrix $\mathbf{R}$ is the range computation following the ray given by the angles α_i and β_j. Figure 4 shows these variables as well as coordinate frames for the map, $(\mathbf{XYZ})^M$, and for the sensor, $(\mathbf{XYZ})^s$.

The range observation model has the following inputs:

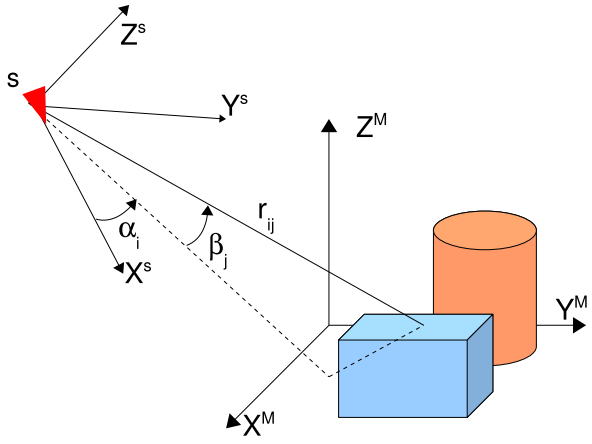

Fig. 4. Model frame, sensor frame, angles α_i and β_j, and the output ranges r_{ij}.

- A 3D geometric model, $\mathcal{M}$.
- A set of sensor parameters: horizontal and vertical angular apertures, Δ_α and Δ_β, horizontal and vertical angular accuracies, δ_α and δ_β, the size of the range matrix, $n_\alpha \times n_\beta$, and range limits, r_{min}, r_{max}.
- A sensor position, $X_s^M = (x_s^M, y_s^M, z_s^M, \theta_s^M, \phi_s^M, \psi_s^M)$.

The operations to execute in order to compute ranges r_{ij} are:

1. Set the projection to view the scene.
2. Set the rendering window size.
3. Render the scene from X_s^M.
4. Read the depth buffer of the graphics card and compute ranges r_{ij}.

Set the Projection. Before using the openGL rendering, the projection parameters have to be set. These parameters are the vertical aperture of the scene view, which is directly the vertical aperture of the modelled sensor, Δ_β, an image aspect ratio ρ, and two parameters limiting the viewing volume with two planes placed at z_N (near plane) and z_F (far plane)[1]. These two last parameters coincide respectively with r_{min} and r_{max} of the modelled sensor. The only parameter to be computed at this step is the aspect ratio ρ. To do this, first the metric dimensions of the image plane, width, w, and height, h, have to be found. The aspect ratio will be derived from them:

$$w = 2r_{min}tg(\frac{\Delta_\alpha}{2}); h = 2r_{min}tg(\frac{\Delta_\beta}{2});\ \rho = \frac{w}{h}; \tag{3}$$

Figure 5 depicts the horizontal cross section of the projection with the associated parameters. The vertical cross section is analogous to the horizontal one.

[1] z_N and z_F are openGL depth values defined in the screen space.

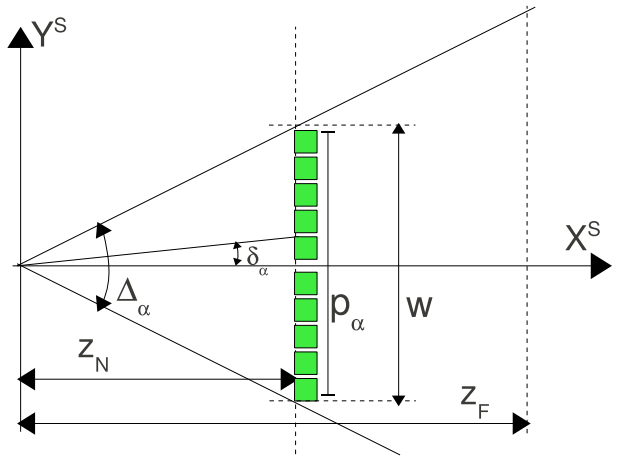

Fig. 5. Horizontal cross section of the projection with the involved parameters. Green squares represent the pixels at the image plane.

Set the rendering window size. Before rendering a 3D scene, the size of the image has to be set. Choosing a size as small as possible is a key issue to speed up the proposed algorithm. Since range sensors have limited angular accuracy, we use that to limit the size of image, in order to avoid renderizing more pixels than those required. Given a sensor with angular accuracies δ_α and δ_β, pixel dimensions of the rendering window are to:

$$p_\alpha = (int)2\frac{tg(\Delta_\alpha/2)}{tg(\delta_\alpha)};\ p_\beta = (int)2\frac{tg(\Delta_\beta/2)}{tg(\delta_\beta)}; \tag{4}$$

Figure 5 shows an horizontal cross section of the projection and the related variables to compute the horizontal pixel size of the rendering window (the vertical pixel size is found analogously).

Render the scene. The scene is renderized from the view point situated at sensor position X_s^M. Beyond computing the color for each pixel of the rendering window, the graphics card also associates to each one a depth value. Moreover, graphics card are optimized to discard parts of the model escaping from the scene, thus having limited the rendering window size and volume speeds up the rendering step. Renderization can be execute in a hidden window.

Read the depth buffer. Depth values of each pixel are stored in the *depth buffer* of the graphics card. They can be read by means of an openGL function that returns data in a matrix $\mathbf{B}$ of size $p_\alpha \times p_\beta$, which is greater in size than the desired matrix $\mathbf{R}$. Read depth values, b_{kl}, are a normalized version of the renderized depth for each pixel. To obtain the desired ranges, we first have to compute the $\mathbf{D}$ matrix, which holds the non-normalized depth values, that is the depth value

of the pixels following the $\mathbf{X}^s$ direction:

$$k = (int)(\frac{1}{2} - \frac{tg(\alpha_i)}{2tg(\frac{\Delta_\alpha}{2})})p_\alpha; \quad l = (int)(\frac{1}{2} - \frac{tg(\beta_j)}{2tg(\frac{\Delta_\beta}{2})})p_\beta$$
$$d_{ij} = \frac{r_{min} r_{max}}{(r_{max} - b_{kl})(r_{max} - r_{min})}; \tag{5}$$

The last equation undoes the normalization computed by the graphics card to store the depth values. The $\mathbf{D}$ matrix has $n_\alpha \times n_\beta$ size, since we compute d_{ij} only for the pixels of interest. Finally, with basic trigonometry we can calculate the desired r_{ij} as:

$$r_{ij} = \frac{d_{ij}}{cos(\alpha_i)cos(\beta_j)} \tag{6}$$

Figure 6 shows the variables involved on this last step, showing the meaning of the d_{ij} and r_{ij} distances in an horizontal cross section of the scene. Equation 6 presents numerical problems when α_i or β_j get close to $\pi/2$. This will limit the aperture of our sensor model. However, section 5 explains how to overcome this limitation when modeling a real sensor with a wide aperture.

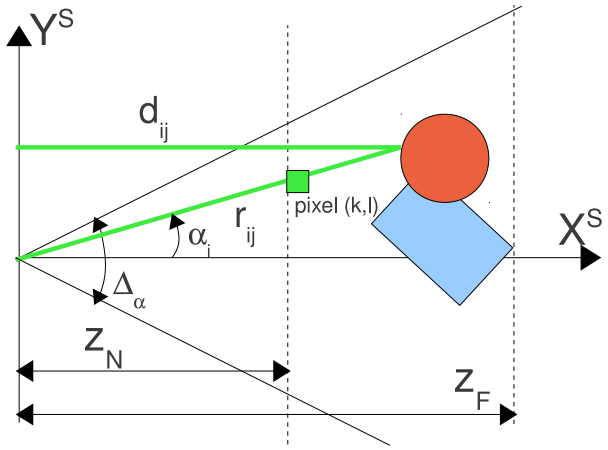

Fig. 6. Horizontal cross section of the projection, with distance d_{ij} and range r_{ij} of the corresponding kl^{th} pixel.

The overall procedure is outlined in algorithm 4. Inside the `for` loops, variables k and l are directly functions of i and j respectively, so we can precompute expressions in equation 5 for k and l, and store values in a vector.

5 Experimental Evaluation

This section presents some results that evaluate the performance of our algorithms. Two range models are presented, corresponding to real laser scanners,

Algorithm 4 Range Sensor Model

INPUT: $\mathcal{M}, \Delta_\alpha, \Delta_\beta, \delta_\alpha, \delta_\beta, n_\alpha, n_\beta, r_{max}, r_{min}, X_s^M$
OUTPUT: $\mathbf{R}$

$w = 2r_{min}tg(\frac{\Delta_\alpha}{2})$; $h = 2r_{min}tg(\frac{\Delta_\beta}{2})$; $\rho = \frac{w}{h}$;
$glSetProjection(\Delta_\beta, \rho, r_{min}, r_{max})$;//rendering volume: vertical aperture, aspect ratio, depth limits
$p_\alpha = (int)2\frac{tg(\Delta_\alpha/2)}{tg(\delta_\alpha)}$; $p_\beta = (int)2\frac{tg(\Delta_\beta/2)}{tg(\delta_\beta)}$;
$glSetWindowSize(0, 0, p_\alpha, p_\beta)$;
$glRenderUpdate(\mathcal{M}, X_s^M)$; //renders the model from the sensor position
$\mathbf{B} = glReadZbuffer(ALL\_IMAGE)$; //reads normalized depth values
for $i = 1..n_\alpha$ **do**
  $\alpha_i = \Delta_\alpha(0.5 - \frac{i}{n_\alpha})$;
  $k = (int)(0.5 - \frac{tg(\alpha_i)}{2tg(\Delta_\alpha/2)})p_\alpha$;
  for $j = 1..n_\beta$ **do**
    $\beta_j = \Delta_\beta(0.5 - \frac{j}{n_\beta})$;
    $l = (int)(0.5 - \frac{tg(\beta_j)}{2tg(\Delta_\beta/2)})p_\beta$;
    $d_{ij} = \frac{r_{min}r_{max}}{(r_{max}-b_{kl})(r_{max}-r_{min})}$;
    $r_{ij} = \frac{d_{ij}}{cos(\alpha_i)cos(\phi_j)}$;
  end for
end for
return $\mathbf{R}$;

and computational time is provided for a testbench 3D scene. Finally, we briefly describe the successful use of all presented algorithms for real-time map-based localization.

5.1 Laser scanner models

Two kind of laser scanner models has been used, using the same software. Table 1 summarizes the input parameters of these laser scanner models. Our implementation sets angular accuracies equal to angular resolutions. Please note also that, due to application requirements, we only model part of the scan provided by the Hokuyo laser.

Table 1. Input parameters of the laser scanner models

Input Parameter	Leuze RS4	Hokuyo UTM 30-LX (partial)
$\Delta_\alpha,\ \Delta_\beta$	190°, 1°	60°, 1°
$n_\alpha,\ n_\beta$	133, 5 points	241, 5 points
$\delta_\alpha = \Delta_\alpha/n_\alpha,\ \delta_\beta = \Delta_\beta/n_\beta$	1.43°, 0.2°	0.25°, 0.2°
$r_{min},\ r_{max}$	0.3, 20 m	0.3, 20 m

Table 2 outlines the derived parameters of the models. Leuze device has an horizontal aperture greater than 180° and that poses numerical problems on

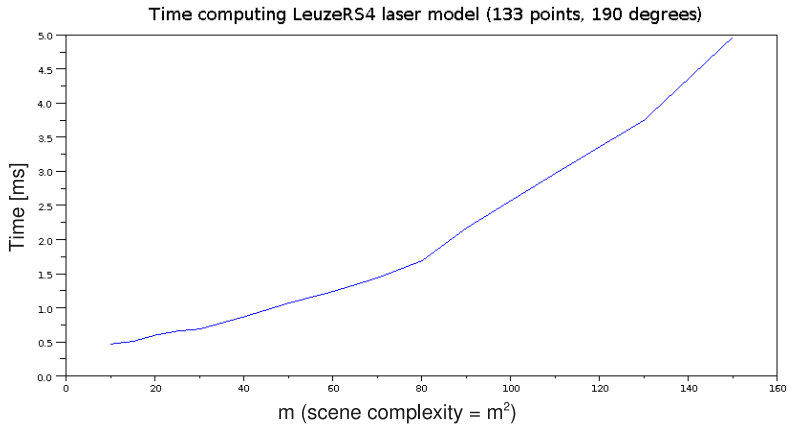

Fig. 7. Time performance versus scene complexity for the Leuze RS4 laser scanner. The sphere object has m^2 elements.

computing equation 6. This issue is overcome by dividing the computation in two scanning sectors, each one with the half of sensor's aperture, so the parameters given in table 2 in the Leuze column are for a single scanning sector.

Table 2. Derived parameters of the laser scanner models

Derived Parameter	Leuze RS4 (per scanning sector)	Hokuyo UTM 30-LX (partial)
w	0.655 m	0.346 m
h	0.005 m	0.005 m
ρ	125	66
p_α	88 pixels	265 pixels
p_β	5 pixels	5 pixels

To evaluate the computational perfomance of the proposed implementation while increasing the scene complexity, we have done a set of experiments consisting on computing 100 times the Leuze model against a testbench environment composed of a single sphere, while increasing the number of sectors and slices of that shape. The results are shown in figure 7. For a given m, the sphere is formed by m sectors and m slices, and thus the scene has m^2 elements.

Please note that using the same software implementation, other range models of devices providing point clouds such as time-of-flight cameras or 3D laser scanners can be easily configured and computed.

5.2 Map-based localization

Gravity constraints and laser scanner models have been used for 3D, real-time, map-based localization of a mobile platform while it navigates autonomously on the UPC campus area introduced in section 2. The mobile platform is a two-wheeled self-balancing Segway RMP200, equiped with two laser devices scanning over the horizontal plane forward and backward (Leuze RS4), and a third laser device (Hokuyo UTM 30-LX) scanning the vertical plane in front of the robot. A particle filter integrates data from these three scanners and from the platform encoders and inclinometers to output a position estimate. At each iteration of the filter, for each particle, height and roll constraints are calculated at the propagation phase by means of the grid versions of gravity constraints, so a negligible time is spent during online executions. On the other hand, expected range observations are computed online using algorithm 4. The filter runs on a DELL XPS-1313 laptop at 5 Hz, using 50 particles. This implies that the computer was calculating $5 \times 50 \times (133+133+241)$ rays per second. The approach has been proved successful as will be documented in future publications.

6 Conclusions

Although efficient computation of 3D range observation models is commonplace in robotics for a wide range of applications, little effort has been put on documenting algorithms to solve this issue. This paper details a set of algorithms for fast computation of range data from 3D geometric models of a given environment using the very well-known openGL programming library. Additionally, we show that the same principles can be applied to the computation of physical constraints of terrestrial mobile platforms and demonstrate our approach for a computationally expensive, real-time application.

References

1. Friedmann, M., Petersen, K., von Stryk, O.: Simulation of Multi-Robot Teams with Flexible Level of Detail. In: Carpin, S., Noda, I., Pagello, E., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2008. LNAI, vol. 5325. 2008.
2. Laue, T., Spiess, K., Rfer, T.: SimRobot - A General Physical Robot Simulator and Its Application in RoboCup. In: A. Bredenfeld, A. Jacoff, I. Noda, Y. Takahashi (Eds.), RoboCup 2005: Robot Soccer World Cup IX, LNAI, No. 4020. 2006.
3. Levinson, J., Montemerlo, M., Thrun, S.:Map-Based Precision Vehicle Localization in Urban Environments. In: Proceedings of the Robotics: Science and Systems Conference. Atlanta, USA. June, 2007.
4. Michel, O.: Cyberbotics Ltd - WebotsTM: Professional Mobile Robot Simulation, In: International Journal of Advanced Robotic Systems, Vol. 1, Num. 1. 2004.
5. Nuske, S., Roberts, J., Wyeth, G.:Robust Outdoor Visual Localization Using a Three-Dimensional-Edge Map. In: Journal of Field Robotics, Num. 26, Vol. 9. 2009.
6. *OBJ* file format, `http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/`
7. *OpenGL*, `http://www.opengl.org`
8. Thrun, S., Fox, D., Burgard, W., Dellaert, F.:Robust Monte Carlo localization for mobile robots. In: Artificial Intelligence, vol. 128. 2001.